# Thomas Jones – CS5567-0002 – Project 3

Quick Note: All these were run on a Mac Studio M1 using MPS support.

## Object Detection with YOLO8

### Training

A number of different hyperparameters were tested, 6 combinations in all, for training this network. The maP50-95 results for each of the networks were similar. The best results came after training the network for 60 epochs using a batch size of 128 with the Adam optimizer with a learning rate of 0.001. and a dropout of 0.3. Training took around 4 hours.

The confusion matrix of the classifications is shown below:

A graph of speed and speed

Description automatically generated

This gave the following confidence curves.

A graph of a speed line

Description automatically generated with medium confidence

A graph of different colors and numbers

Description automatically generated

A graph of a graph

Description automatically generated with medium confidenceA graph of different colors and sizes

Description automatically generated

Training followed typical paths of reduction of loss and increase in accuracy over time.

A group of graphs showing loss

Description automatically generated with medium confidence

A graph of a graph of a training

Description automatically generated with medium confidence

The final set of test images show correct detection of both speed limit signs and street lights. Only in the case of the red light in the lower left corner was the confidence low and the 50 KPH speed limit sign was mis-classified as 90. The image in the lower left is unreadable but was classified as KPH 100. If this was being used for actual speed control this would be problematic.

A screenshot of a video camera

Description automatically generated

### Detection Discussion

There was no obvious difference between video detection and image detection. However, qualitatively observing the different test scenarios, it is clear that the video uses artificial data and hence is somewhat cleaner in terms of background noise.

Partially visible or unreadable objects that still maintained the correct shape were detected but the specific “type” of the object, i.e. 90 vs 50, were misclassified in the tests. This is likely due because while the YOLO network is good at detecting “gross” shapes, i.e. identifying stop signs vs speed limit signs, the detail of the numbers within the signs is problematic. A secondary network would be recommended, perhaps a simpler CNN, that would be used to read the actual limit when a speed limit sign was detected, i.e. a post classifier classifier. Where color was involved as with red light vs green light, the channel differences were used to separate those two classes.

## Segmentation Results

### UUNet

Due to upgrades for Tensorflow the error function needed to be slightly rewritten to no longer use deprecated methods. In all 5 different configurations were tested (see spreadsheet) with most results meeting expected results. It was found that batch size adjustments had the most impact with the best DICE score being 0.902 or a 90.2% coverage match vs the test masks. This was achieved with a batch size of 15 over a total of 100 epochs though the best epoch against validation loss was at 93.

A graph of training and validation

Description automatically generated

This produced the following test images (only images with highlighted areas shown for brevity)

A white mask on a black background

Description automatically generated

A white mask on a black background

Description automatically generated

A white mask on a black background

Description automatically generated

A white mask on a black background

Description automatically generated

A white mask on a black background

Description automatically generated

A white outline of a mask

Description automatically generated

### SegFormer

The SegFormer image segmentation ended up being challenging for some deep technical issues with Pytorch, specifically M1 does not support float64 so results were inconsistent with CPU or CUDA results. See <https://github.com/Lightning-AI/pytorch-lightning/discussions/15407> for a quick reference. As a result these were run on the CPU which then produced the correct results. While a this was a difficulty for the experimenter (who could have used Colab) it does speak to some of the broader consumer challenges that might be faced in the future as well as for future experimenters.

Three random images were selected, shown with corresponding masks.

A bathroom with a sink and a window

Description automatically generated

A bathroom with a tub sink and a window

Description automatically generated

Comparing the data against the mask:

A graph of a fish and a hook

Description automatically generated with medium confidence

We can see that the results reasonably match the masks provided. The fine tuning produced a mean IOU of 0.530, a mean accuracy of 0.818, and an overall accuracy of 0.982.

### Segmentation Discussion

While similar in appearance the two networks performed different tasks in different manners. A big constraint on direct comparison was the limited size of the SegFormer dataset. Without this effective parameter changes were not feasible so the observation that UUNet, for the image set given, performed better with smaller batches, cannot be compared. In terms of geometric structures, SegFormer appears to be a better choice against non-organic or more regular shapes. UUNet operated well (higher metrics) against a task that was more structure filling in nature, i.e. filling out the area of a possible tumor, hence a more organic shape. Neither system seemed to have any issues with overlapping structures. Both networks were trained for 100 epochs using the Adam optimizer with default parameters (lr=0.002, etc).

## Transfer Learning

The same additional architecture was used for both networks, specifically a resizing layer at the head and the addition of a batch normalization and dense layer at the tail. The activation function in both cases was switched to softmax from relu. Additionally, initial experimentation found that tanh worked better as an activation for the dense layer, it is possible this helped to exclude labels where two labels were close though further experimentation would be necessary.

### ResNet50

The training performance showed an expected initial sharp uptick in accuracy with a tailing plateau. The validation accuracy followed a similar path eventually converging.

A graph of a graph

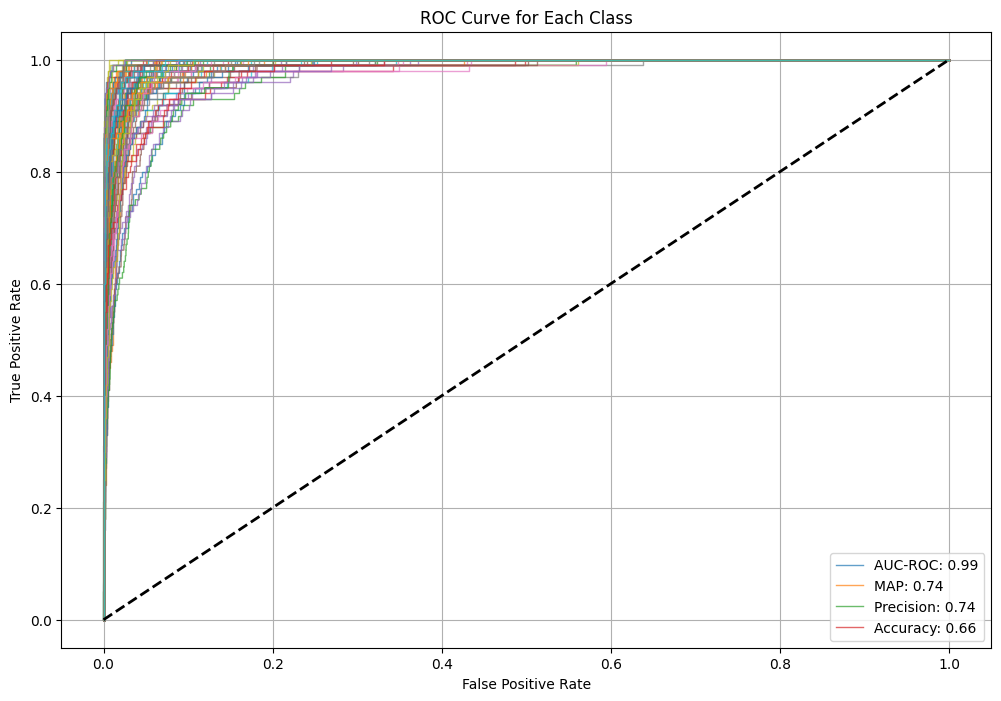
Description automatically generated

The loss curve shows similar results though as expected, training loss remained well above validation loss which remained relatively stable. While this might be an indicator of overfitting the convergence of the validation accuracy and lack of subsequent drop indicates a local minima was likely reached and little further training helped.

A graph of training and validation

Description automatically generated

The ROC curves for each of the classes show that the classification overall performed well given that the original images are only 32x32.



Looking at a few actual results,

A collage of different animals

Description automatically generated

It is at least visually clear that, while there were misses, the misses are actually reasonable mistakes. It is possible, for example, that the first image *could*be a road and the cloud does look like a sea. The butterfly, at this resolution, does appear skunk-like as well.

### VGG19

The accuracy characteristics for this network show what we would normally expect from training a network with a clear gap between training and validation accuracy.

A graph of a graph

Description automatically generated

While the loss curves to indicate an increase in validation error (though not necessarily shown in the accuracy) as training continued. While not completely indicative of overtraining it is clear that fewer epochs might have been required to achieve the results.

A graph of training and validation

Description automatically generated

The ROC curves for the various classes indicate accurate classification considering the resolution of the source images.

A graph of a curve

Description automatically generated

The predicted test labels are reasonable considering the resolution of the images. Apple was missed though it’s possible that was overtraining. Without further experiments that cannot be determined. The network clearly has issues with animals, i.e. more organic shapes, but even a human may not be able to identify the butterfly at this resolution.

A collage of images of animals

Description automatically generated

### Discussion

ResNet, for the hyperparameters used, performed slightly better than VGG and would be expected in this case as ResNet employs skip connections to better deal with vanishing gradients, i.e. ResNet is a better head.

## Final Observations

My biggest observation is that the networks performed best when they were task specific and the task at hand is what determined complexity of the network necessary to solve the task. There is no “one AI to rule them all” per the No Free Lunch theorem. Careful understanding of the data is required to select the proper tool to handle that data. For example, while YOLO is clearly performant in terms of general object detection and identification the lower level details require a different solution. The segmentation tasks show that different architectures are best for handling organic vs non-organic shapes. While not shown in these experiments, it is unlikely that SegFormer would do well mapping out tumors. This might be the difference in the domains of the tasks, specifically multi-class segmentation vs single class but this would require additional testing to verify.

The image classification networks demonstrated the novel approaches (skip connections) that are required to overcome problems that are structurally present in every deep network, in this case the vanishing gradient problem. Different overall topologies that are not linear stacks of layers will be necessary for more generalized solutions, i.e. Hopfield/Boltzmann networks where interconnectedness is greater.

## Appendix – Code

### YOLO Changes

Most of the changes were fixes to the paths of the datafiles, i.e. of the form:

Image\_dir = './datasets/cardetection/car/train/images'

For the different training runs:

# Build from YAML and transfer weights

Final\_model = YOLO('yolov8n.pt')

# Training The Final Model

# Result\_Final\_model = Final\_model.train(data="./cardetection/car/data.yaml",epochs = 2, batch = 128, optimizer = 'sgd', lr0=0.1)

# Result\_Final\_model = Final\_model.train(data="./cardetection/car/data.yaml",epochs = 20, batch = 128, optimizer = 'sgd', lr0=0.01, device='mps')

# Result\_Final\_model = Final\_model.train(data="./cardetection/car/data.yaml",epochs = 20, batch = 128, optimizer = 'adam', lr0=0.001, device='mps')

# Result\_Final\_model = Final\_model.train(data="./cardetection/car/data.yaml",epochs = 40, batch = 128, optimizer = 'adam', lr0=0.001, device='mps')

# Result\_Final\_model = Final\_model.train(data="./cardetection/car/data.yaml",epochs = 40, batch = 128, optimizer = 'adam', lr0=0.004, device='mps')

Result\_Final\_model = Final\_model.train(data="./cardetection/car/data.yaml",epochs = 60, batch = 64, optimizer = 'adam', lr0=0.002, dropout=0.3, device='mps')

### UUnet Changes

Due to Tensorflow library upgrades the loss functions were rewritten to no longer use deprecated methods.

# function to create dice coefficient

def dice\_coef(y\_true, y\_pred, smooth=100):

y\_true\_flatten = tf.reshape(y\_true, [-1])

y\_pred\_flatten = tf.reshape(y\_pred, [-1])

intersection = tf.reduce\_sum(y\_true\_flatten \* y\_pred\_flatten)

union = tf.reduce\_sum(y\_true\_flatten) + tf.reduce\_sum(y\_pred\_flatten)

return (2 \* intersection + smooth) / (union + smooth)

# function to create dice loss

def dice\_loss(y\_true, y\_pred, smooth=100):

return -dice\_coef(y\_true, y\_pred, smooth)

# function to create iou coefficient

def iou\_coef(y\_true, y\_pred, smooth=100):

intersection = tf.reduce\_sum(y\_true \* y\_pred)

sum = tf.reduce\_sum(y\_true + y\_pred)

iou = (intersection + smooth) / (sum - intersection + smooth)

return iou

The model checkpoint stored the file using the .keras format

callbacks = [ModelCheckpoint('unet.keras', verbose=1, save\_best\_only=True)]

And the best network was then loaded before testing

model.load\_weights('unet.keras')

### SegFormer Changes

The primary changes where to select 3 random images for testing

#pick 3 random images from the training set

import random

img1, img2, img3 = random.sample(range(1,len(train\_dataset)+1), 3)

img1Str = str(img1).rjust(8, '0')

img2Str = str(img2).rjust(8, '0')

img3Str = str(img3).rjust(8, '0')

image1 = Image.open(f'./ADE20k\_toy\_dataset/images/training/ADE\_train\_{img1Str}.jpg')

image2 = Image.open(f'./ADE20k\_toy\_dataset/images/training/ADE\_train\_{img2Str}.jpg')

image3 = Image.open(f'./ADE20k\_toy\_dataset/images/training/ADE\_train\_{img3Str}.jpg')

#show the images

import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

axs[0].imshow(image1)

axs[1].imshow(image2)

axs[2].imshow(image3)

plt.show()

import matplotlib.pyplot as plt

import numpy as np

def show\_segmentation\_map(predicted\_segmentation\_map, image, ax):

color\_seg = np.zeros((predicted\_segmentation\_map.shape[0],

predicted\_segmentation\_map.shape[1], 3), dtype=np.uint8) # height, width, 3

palette = np.array(ade\_palette())

for label, color in enumerate(palette):

color\_seg[predicted\_segmentation\_map == label, :] = color

# Convert to BGR

color\_seg = color\_seg[..., ::-1]

# Show image + mask

img = np.array(image) \* 0.5 + color\_seg \* 0.5

img = img.astype(np.uint8)

ax.imshow(img)

# plt.show()

return img

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

img1 = show\_segmentation\_map(predicted\_segmentation\_map1, image1, axs[0])

img2 = show\_segmentation\_map(predicted\_segmentation\_map2, image2, axs[1])

img3 = show\_segmentation\_map(predicted\_segmentation\_map3, image3, axs[2])

plt.show()

### ResNet50 Changes

Changes in model construction

# Define your model

model = tf.keras.Sequential() # Resize layer

model.add(tf.keras.layers.Resizing(224, 224, interpolation="bilinear"))

model.add(res50\_model) # Add resnet50

model.add(Flatten()) # Flatten the output and add dense layers

model.add(tf.keras.layers.BatchNormalization())

model.add(Dense(512, activation='tanh'))

##Note, just having Softmax seems to do it

model.add(Dense(100, activation='softmax')) # Final layer for 100 classes

Optimizer

optimizer = tf.keras.optimizers.Adam()

model.compile(optimizer= optimizer,

loss='categorical\_crossentropy',

metrics=['accuracy'])

Fit call

history = model.fit(

# train\_datagen.flow(x\_train, y\_train, batch\_size = 64),

# validation\_data = val\_datagen.flow(x\_val,y\_val, batch\_size = 64),

train\_datagen.flow(x\_train, y\_train),

validation\_data = val\_datagen.flow(x\_val,y\_val),

epochs = 100,

verbose = 1,

# callbacks = [learning\_rate\_reduction]

)

### VGG Changes

Changes in model construction

# Define your model

model = tf.keras.Sequential() # Resize layer

model.add(tf.keras.layers.Resizing(224, 224, interpolation="bilinear"))

model.add(res50\_model) # Add resnet50

model.add(Flatten()) # Flatten the output and add dense layers

model.add(tf.keras.layers.BatchNormalization())

model.add(Dense(512, activation='tanh'))

##Note, just having Softmax seems to do it

model.add(Dense(100, activation='softmax')) # Final layer for 100 classes

Optimizer

optimizer = tf.keras.optimizers.Adam()

model.compile(optimizer= optimizer,

loss='categorical\_crossentropy',

metrics=['accuracy'])

Fit call

history = model.fit(

# train\_datagen.flow(x\_train, y\_train, batch\_size = 64),

# validation\_data = val\_datagen.flow(x\_val,y\_val, batch\_size = 64),

train\_datagen.flow(x\_train, y\_train),

validation\_data = val\_datagen.flow(x\_val,y\_val),

epochs = 100,

verbose = 1,

# callbacks = [learning\_rate\_reduction]

)